01 May 2003

Metrics and Models for Cost and Quality of Component-Based Software

Sahra Sedigh
Missouri University of Science and Technology, sedighs@mst.edu

Arif Ghafoor

Raymond A. Paul

Metrics and Models for Cost and Quality of Component-Based Software

Sahra Sedigh-Ali and Arif Ghafoor
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907
{sedigh, ghafoor}@ecn.purdue.edu

Raymond A. Paul
Department of Defense, OASD/C3I
Pentagon
Washington, DC 20543
ray.paul@oasd.pentagon.mil

Abstract

Quality and risk concerns currently limit the application of commercial off-the-shelf (COTS) software components to non-critical applications. Software metrics can quantify factors contributing to the overall quality of a component-based system, and models for tradeoffs between cost and various aspects of quality can guide quality and risk management by identifying and eliminating sources of risk. This paper discusses metrics and models that can be used to alleviate quality concerns for COTS-based systems, enabling the use of COTS components in a broader range of applications.

1. Introduction

The paradigm shift to *commercial off-the-shelf* (COTS) components appears inevitable, necessitating drastic changes to current software development and business practices. Quality and risk concerns currently limit the application of *COTS-Based Systems* (CBSs) to non-critical applications. New approaches to quality and risk management are needed to handle the growth of CBSs [4, 6]. With software development proceeding at unprecedented speed, in-house development of all system components may prove too costly in terms of both time and money, as witnessed by the outsourcing trend currently present in commercial software development. Large-scale component reuse and COTS component acquisition can generate savings in development resources, which can then be applied to quality improvement, such as enhancements to reliability, availability, and ease of maintenance.

From a historical perspective, software failures in both traditional and component-based systems have had drastic consequences for the organizations employing these systems. Software errors in the automated baggage control system of the Denver International Airport delayed the opening 16 months, dramatically increasing the construction costs [10]. Within the Department of Defense, where software is subject to rigorous testing and debugging, a software error caused severe failure in a mission involving the Clementine satellite. This satellite was launched into lunar orbit in the spring of 1994, and one objective of its launch was the testing of targeting software that may have later been used in missile defense systems [10]. If the failure had gone undetected, and the software system had been used in a missile defense system, such a failure may have had calamitous consequences.

Such case studies serve as important stepping-stones on the path to component-based software development, as many of the same problems threaten CBSs. Quality and performance concerns can be alleviated by using *software metrics* to guide quality and risk management in a CBS, accurately quantifying various factors contributing to the overall quality of a CBS, and identifying and eliminating sources of risk. Metrics can guide decisions throughout the software life cycle and determine whether software quality improvement initiatives are financially worthwhile [28, 21, 20].

As in any development or manufacturing process, software quality is achieved at a cost. This paper outlines existing challenges in cost and quality management of CBSs and illustrates the use of metrics for quantifying the concept of software quality, investigating the tradeoff between cost and quality, and using the information gained to guide quality management.

2. Software Metrics for CBSs

Measurement enables engineers to quantify product quality and performance, to evaluate the development process and product attributes that impact quality and performance, and to demonstrate how product and process changes impact these attributes. A proper choice of basic metrics combined with a selection of powerful tools and true integration of these metrics and tools forms the foundation of efficient and robust software project management. To this end, a metrics-guided approach to the quality man-

Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)
0-7695-1928-8/03 $17.00 © 2003 IEEE

agement of CBSs can aim to provide software developers and managers of CBSs with the same benefits attained by metrics in developing in-house software. The first step is identifying a set of metrics tailored to the specific needs of CBSs.

Metrics can guide risk and quality management and help to reduce the risks encountered during the planning and execution of software development, resource and effort allocation, scheduling, and product evaluation [18, 21]. Risks can include performance issues, reliability, adaptability, and return on investment. Risk reduction can take many forms, such as using component wrappers or middleware, replacing components, relaxing system requirements, or even issuing legal disclaimers for certain failure-prone software features. Metrics let developers identify and isolate these risks, then take corrective action.

One of the keys to success is selecting appropriate metrics - especially metrics that provide measures applicable over the entire software cycle and address both software processes and products. In choosing metrics, the software engineering environment of both the development and maintenance cycles should be considered, as well as the economics of metrics collection and analysis. To reduce the possibility of having a single bottleneck to measurement accuracy, redundant or cross-related metrics can be defined, where each of the related metrics can validate the data provided by the others. The metrics selected should also have a strong basis in industry or government practice for establishing "rule of thumb" thresholds for use by software managers [11, 28].

An important difference between metrics for CBSs and traditional systems is the unavailability of "size" as a metric. Most traditional metrics sets incorporate the size of the source code, measured in *Lines of Code* (LOC), into several metrics. This size is generally not known for COTS components, hence, if a measure of program or component size is required, alternate measures should be used. One such measure is the number of *use cases* - business tasks the application performs - that a given component supports [26].

CBS metrics also approach *time to market* differently. Component acquisition changes the concept of time to market because developers may not know the component development time and cannot incorporate it into time calculations. For CBSs, a simple delivery rate measure can replace the time to market measure. One proposed measure divides the number of use cases by the elapsed time in months [26].

Another challenge arises from the fact that the thorough component level testing required for metrics collection may not be possible, due to inaccessibility of the component source code. Test results reported by the component development team may be of limited use, as integration testing is performed at the time of deployment, and discrepancies may arise between the metrics collected at system and component levels.

An initial set of system level metrics for CBSs is described in 1. These metrics are intended to help software developers and managers select appropriate components from a repository of software products and aid in deciding between using COTS components or developing new components. The primary considerations are cost, time to market, and product quality.

Of the metrics defined in Table 1, a subset is of particular importance to CBS development and design. This subset, which appears italicized in Table 1, includes seven metrics: cost, time to market, software engineering environment, complexity, test coverage (either integration or end-to-end), and reliability. A study of the literature, including industrial case studies [11, 28, 25, 24, 2], underlines the importance of this subset.

Cost is an obvious choice for the focus group. Time to market is closely related to cost and directly impacts the market viability of the software product. The software engineering environment metric is of interest due to its potential for predicting quality software. From the quality perspective, the focus is on the test coverage and reliability metrics. Complexity has been chosen, as it impacts the robustness of the software, as well the cost, as detailed later in this paper. Reliability is currently the bottleneck for CBS development, and is hindering the deployment of CBSs in critical applications [19, 17, 27]. The test coverage metric is of importance, as it is closely related to reliability and provides an automated means for measuring which code has been tested.

This subset of metrics satisfies the criteria mentioned above for metrics selection, as the six metrics are relevant to the software engineering environment of CBS development and maintenance, can be collected in a cost-effective manner, are cross-related to each other, and can be used to establish rule-of-thumb guidelines for software management.

Because the metrics are interdependent, understanding the relationships between them can aid decision-making regarding CBS quality-improvement investments. The most obvious relationship is between cost and the quality metrics, such as reliability [7, 13]. However, more subtle relationships exist, such as those among time to market, test coverage, and reliability. Delayed product release because of testing and debugging can result in reduced revenues or, in extreme cases, loss of the market to a competitor with an earlier release. On the other hand, premature product release can lead to lower reliability. Understanding the relationships among time to market, test coverage, and reliability can help in selecting a suitable release schedule [9]. Another effective strategy involves using the software engineering environment in conjunction with the quality metrics to encourage vendors to improve their software development process and adhere to standards, thus increasing the

Table 1. System-level Software Engineering Metrics for CBSs.

Category	Metric	Evaluates/Measures
Management	*Cost*	Total software development expenditure, including costs of component acquisition, integration, and quality improvement
	Time-to-market	Elapsed time between development start and component acquisition to software delivery
	Software Engineering Environment	Capability and maturity of the development environment
	System Resource Utilization	Utilization of target computer resources as a percentage of total capacity
Requirements	Requirements Conformance	Adherence of integrated product to defined requirements at various levels of software development and integration
	Requirements Stability	Level of changes to established software requirements
Quality	Adaptability	Integrated system's ability to adapt to requirements changes
	Complexity	Component interface and middleware or integration code complexity, number of components
	Integration Test Coverage	Fraction of the system that has undergone satisfactory integration testing
	End-to-End (E2E) Test Coverage	Fraction of the system that has undergone satisfactory E2E testing
	Fault Profiles	Cumulative number of detected faults
	Reliability	Probability of failure-free system operation over a specified period of time
	Customer Satisfaction	Degree to which the software meets customer expectations and requirements

likelihood that users will select their component.

One possible approach to modeling the relationships among the metrics is an influence diagram [22]. An *influence diagram* is a network for probabilistic and decision analysis models. The nodes correspond to variables that can be constants, uncertain quantities, decisions, or objectives. The arcs reveal the probabilistic dependence between the uncertain quantities and the information available at the time of the decisions. Detailed data about the variables is stored within the nodes, so the program graph is compact and focuses attention on the relationships among the variables. The flexibility, tractability, graphic nature, and intuitiveness of influence diagrams make them an attractive choice. To construct the influence diagrams, and to determine the initial metrics values, data from case studies, field tests, and simulation can be used. A sample influence diagram is presented in Figure 1, and depicts the relationships among the focus subset of the metrics. In this figure, the rectangles, ovals, and rounded rectangle represent decisions, uncertain quantities, and the objective value, respectively.

Beginning at the rightmost node of Figure 1, the objective value being maximized is the software value, which can be expressed in terms of the overall return on investment, the market share, or other common measures of product value. The decisions to be made involve the time to market and reliability of the product. For instance, how long to continue testing and debugging before releasing the product, which in turn determines how quickly the product will be released, and how reliable it will be. These decisions directly impact the software value, as an early product release will lead to a competitive edge, and a possible increase in market share, and higher reliability will make the product more desirable, and hence, more valuable.

The uncertain quantities driving the decisions are cost, software engineering environment, and test coverage, as well as the time to market and reliability. The software engineering environment influences all of the other uncertain quantities, as a better software engineering environment will achieve higher test coverage, and hence higher reliability within lower cost and shorter time to market. Test coverage influences reliability, as higher test coverage is more likely to remove a greater number of software faults, leading to a lower failure rate and higher reliability. Time to market influences cost, as the a late release may be costly in terms of increased testing costs, as well as loss of rev-

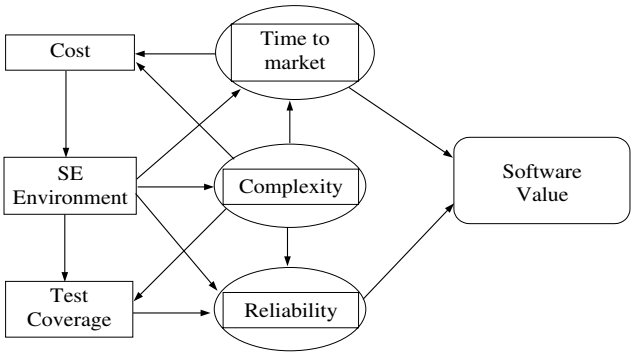

Figure 1. Sample influence diagram for a subset of the software metrics

enue due to losing market share to competitors. Reliability influences time to market, as a more reliable product can be released more quickly.

Complexity influences test coverage and reliability, as overly complex software is less likely to be tested thoroughly and will be less reliable as a result. Complexity also influences time to market, as testing will be more time consuming. For reasons described later in this paper, complexity is also affects the cost of the software. Having a clear picture of these influences and interrelationships facilitates decision-making regarding investments in quality improvement initiatives.

3. Modeling Cost and Quality in CBSs

In deciding between in-house development and COTS component acquisition, the anticipated effect on system quality is an important concern. Software quality can be measured from several different perspectives, including the level of satisfaction of the customer, the key attributes of the software, or freedom from defects in the software's operation. In metrics-guided quality management, software metrics are used to guide the allocation of resources to quality improvement initiatives.

The *cost of quality* (CoQ) represents the resources dedicated to improving the quality of the product being developed. For example, increasing or maintaining reliability incurs costs that can be considered the costs of reliability. The overall CoQ is the sum of such costs plus other costs that cannot be directly attributed to factors measured by the quality metrics. Quality costs, then, represent "the difference between the actual cost of a product or service and what the reduced cost would be if there were no possibility of substandard service, failure of products, or defects in their manufacture [8]."

We concern ourselves with the *cost of software quality* (CoSQ) - corresponding to a portion of the cost metric in defined earlier - which can be divided into two major types: cost of conformance and cost of nonconformance.

The cost of conformance derives from the amount the developer spends on attempts to improve quality. We can further divide conformance costs into prevention and appraisal costs. Projects incur prevention costs during activities targeted at preventing defects, such as training costs, software design reviews, and formal quality inspections. Likewise, activities that involve measuring, evaluating, or auditing products to assure conformance to quality standards and performance incur appraisal costs. These activities include code inspections, testing, quality audits, and software measurement activities such as metrics collection and analysis.

The cost of nonconformance includes all expenses the developer incurs when the system does not operate as specified. Internal failure costs stem from nonconformance occurring before the product ships to the customer, such as the costs of rework in programming, defect management, reinspection, and retesting. External failure costs arise from product failure after delivery to the customer. Examples include technical support, maintenance, remedial upgrades, liability damages, and litigation expenses.

In any development process, models that depict the relationship between costs and quality can guide decisions regarding investments in quality improvement. Discussions of such models in the economics and management literature [15, 12, 8] generally depict a nonlinear relationship between CoQ and quality. Accurate cost-quality models can be invaluable to managers and developers, guiding resource and cost management and other aspects of the software development process.

In [23], the cost of quality and return on quality are evaluated from the perspective of software development. Three metrics are introduced in the software engineering context, CoSQ, *return on software quality* (RoSQ), and *software quality profitability index* (SQPI), which determines whether a particular quality initiative will create value that exceeds its investment. This study also assumes non-CBS software development. Similar composite metrics can be defined for CBSs, and used to model cost-quality tradeoffs in such environments. [16] conducts an assessment of the impact of reuse on quality and productivity in object-oriented systems. The metrics used are size, reusability, effort, productivity, and number of defects. These metrics can be used in CBSs, provided that the notion of size is suitably defined.

Very little, if any, research has been conducted on the economics of quality in CBS development. Cost models for software reuse have been widely studied, but quality is largely ignored in these studies. COCOMO 2.0 [5] takes

software reuse into account, and allows the use of logical lines of code (LLOC) as the standard measure. The authors suggest using the checklist developed by Park at the SEI to explicitly define a LLOC. This model has limited applicability to CBS, as COTS software, libraries, and auto-generated code are excluded when counting the LLOCs. Where possible, COCOMO 2.0 can be used to estimate some component level cost factors.

The *Constructive COTS* (COCOTS) model [1, 3], one of the suites of COCOMO models, can be used to estimate effort and schedule for CBS development. This model is an amalgam of four related sub-models: (1) COTS component assessment, (2) COTS component tailoring, (3) COTS glue code development, and (4) COTS volatility. Seventeen attributes, including correctness, availability/robustness, and security, are also defined as most influential during a final selection assessment of COTS software. The assessment sub-model is intended for use in the initial stages of development, and is aimed at selecting the most suitable COTS component from a set of candidates. COCOTS can currently yield effort estimates only; schedule estimation is yet to be incorporated into the model.

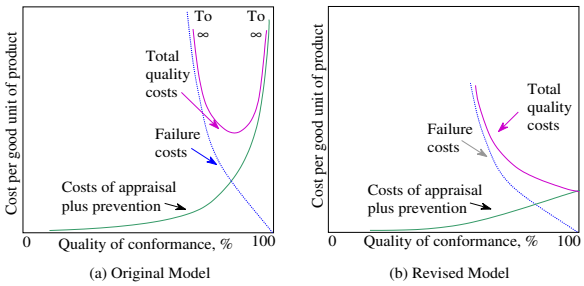

Figure 2. Optimum Quality Costs Model (adapted from [14])

Figure 2(a) depicts the classic model of optimum quality costs. In this model, which shows the relationship between the cost per good unit of product and the quality of conformance, expressed as a percentage of total conformance, prevention and appraisal costs rise asymptotically as the product achieves complete conformance. Recent technological developments inspired a revised model that reflects the ability to achieve very high quality, or "perfection," at finite costs. Shown in Figure 2(b), this model, proposed in [14], has two key concepts. The first is that moderate investments in quality improvement result in a significant decrease in the cost of nonconformance. The second key concept is that focusing on quality improvement by defect prevention results in an overall decrease in the cost of testing and related appraisal tasks.

These models can be analyzed in terms of the metrics defined for a CBS. The quality of conformance in the original model can represent one quality metric, such as test coverage or reliability. Accordingly, the vertical axis represents a CoSQ component - namely, the portion of quality costs dedicated to improving the particular quality factor. Intuitively, the same nonlinear relationship should hold. Increasing the investment in improving a certain quality factor should increase the value of the corresponding metric, and the amount of this increase should taper off as the product achieves high quality levels. As described in previous sections of this paper, it is generally difficult to accurately evaluate and quantify the quality and performance of COTS components. Hence, "perfect" quality may be very difficult to claim, and will not be achievable at finite costs. For these reasons, Figure 2(a) may be a better model for quality costs in CBSs.

In adapting Figure 2(a) to CBSs, we maintain that conformance costs are lower than those of traditional software systems. For CBSs, only black box testing can generally be assumed feasible, so the costs incurred in white box testing are avoided. These savings may be cancelled by the costs incurred during the selection of appropriate components, leading to the conclusion that appraisal costs for a CBS are comparable to appraisal costs in a traditional software system. Prevention costs may be lower for a CBS, as the black box nature of the components limits the possibility of extensive preventative measures. Comparable appraisal costs and lower prevention costs result in an overall lower cost of conformance.

In evaluating the failure costs of a CBS, one should note that the failures occurring later in the software life cycle are costlier than those occurring early in the software life cycle. Generally, internal failure costs incurred due to failures prior to the release of the software, such as defect management, are lower than external failure costs incurred after release, such as defect notification and litigation costs. Due to the limited testing possible for COTS components, in similar software engineering environments, fewer failures will be detected prior to the release of a CBS, as compared to a traditional software system. Fewer detected failures imply lower internal failure costs. If the software being compared is of similar quality (comparable number of failures), this delays the detection of failures to after the release of the software, increasing the cost of such failures, which are now considered external failure costs. Lower internal failure costs and higher external failure costs may lead to comparable overall costs of nonconformance. Considering the lower conformance costs of CBSs, we can generally conclude that the total quality costs of a CBS are lower than a non-component based software system of comparable quality.

Although we may be able to determine the overall CoQ

with reasonable accuracy, determining the amount dedicated to improving a particular quality factor is difficult because all factors interrelate. For quality metrics such as customer satisfaction, the relationship between cost and quality may be too complex for such a simple model, as increased investments in quality improvement may be invisible to the customer. For example, users may find 95 percent reliability satisfactory, making further investments in reliability pointless. Further, customer satisfaction may increase in jumps, resulting in a discontinuous cost-quality curve, although empirical studies should verify this behavior.

Recent work by Abts [2], examines the relationship between the number of COTS components in a CBS, and tradeoffs between maintaining and retiring the system. Abts postulates that increasing the number of COTS components is economically beneficial only up to a certain point, where the savings resulting from the use of COTS components break even with the maintenance costs arising from the volatility of such components. The study recommends lowering the number of COTS components by increasing the *functional density* of the components used, in other words, using *leaner* components. Although the resulting system will benefit from reduced complexity, the disadvantage to using lean components is the lack of robustness that results. As a component is relied upon for providing a greater number of functions to the system, a failure in the component is more costly to the system due to decreased redundancy in component functionality. Quality costs arising from a lack of robustness can be classified as costs of non-conformance.

Conversely, using a greater number of COTS components leads to higher system complexity, increasing not only the maintenance costs mentioned in [2], but also appraisal and prevention costs in general, as testing a complex system is generally lengthy and expensive. A complex system is generally more likely to fail; hence, the costs of nonconformance will also increase in an overly complex system.

This leads to a generalization of the results of [2], whereby quality costs are modeled as having a non-linear relationship with complexity, as depicted in Figure 3. In this view, an optimal value of system complexity will maintain the best balance between leanness and robustness, minimizing quality costs. A decision theoretic approach can be utilized to find the best tradeoff among quality, cost, and complexity, facilitating high quality of conformance at low cost and moderate complexity.

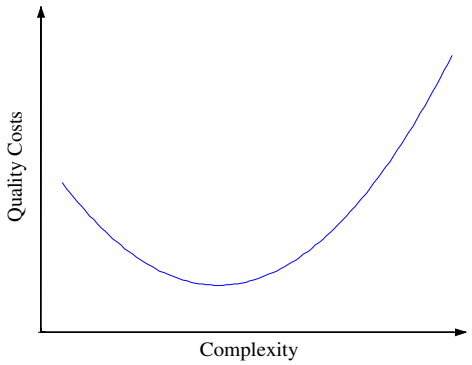

Figure 3. Relationship between Quality Costs and Complexity of a CBS

4. Conclusions

Quality and risk concerns currently limit the application of COTS-based system design to non-critical applications. The cost and quality metrics and models discussed in this paper can aid developers and managers in deciding on optimal quality improvement initiatives for CBSs, as well as analyzing the return on investment in quality improvement initiatives. Research on metrics-guided quality management can enable extensive economic and engineering analyses for CBSs, including identification of cost factors and cost-benefit analysis involving the unique risks associated with CBSs, determination of the complexity and cost associated with integration, interoperability, and middleware development, and estimation of the costs associated with the unique testing requirements of CBSs. The findings can alleviate concerns about the risks associated with deploying COTS components in critical applications, facilitating the use of components in a broader range of applications.

References

[1] C. Abts. COTS software integration cost modeling study. Technical Report USC-CSE-98-520, University of Southern California, Los Angeles, 1998.

[2] C. Abts. COTS-Based Systems (CBS) functional density – a heuristic for better CBS design. In J. Dean and A. Gravel, editors, *Proc. of the 2002 Int'l Conf. on COTS-Based Software Systems (ICCBSS 2002)*, volume 2255 of *Lecture Notes in Computer Science*, pages 1–9. Springer, Feb. 2002.

[3] J. Baik, N. Eickelmann, and C. Abts. Empirical software simulation for COTS glue code development and integration. In *Proc. of the 2001 Int'l Conf. on Computer Software and Applications (COMPSAC 2001)*, pages 297–302, Oct. 2001.

[4] B. Boehm and C. Abts. COTS integration: Plug and pray? *IEEE Computer*, 32(1):135–138, Jan. 1999.

[5] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby. Cost models for future lifecycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1:57–94, 1995.

[6] P. Brereton and D. Budgen. Component-based systems: A classification of issues. *IEEE Computer*, 33(11):54–62, Nov. 2000.

[7] R. Burnett. A trade-off method between cost and reliability. In *Proc. 17th Int'l Conf. Chilean Computer Science Society (SCCC '97)*, 1997.

[8] J. Campanella. *Principles of Quality Costs: Principles, Implementation, and Use*. ASQ Quality Press, Milwaukee, Wis., 3rd edition, 1999.

[9] T. Chàvez. A decision-analytic stopping rule for validation of commercial software systems. *IEEE Tran. on Software Eng.*, 26(9):907–918, Sept. 2000.

[10] W. W. Gibbs. Software's chronic crisis. *Scientific American*, 271(3):72–81, Sept. 1994.

[11] R. B. Grady. Successfully applying software metrics. *IEEE Software*, 11(5):18–25, May 1994.

[12] T. J. Haley. Software process improvement at raytheon. *IEEE Software*, 13(6):33–41, Nov. 1996.

[13] M. E. Helander, M. Zhao, and N. Ohlsson. Planning models for software reliability and cost. *IEEE Trans. on Software Eng.*, 24(6):420–434, Jun. 1998.

[14] J. M. Juran and F. M. Gryna. *Juran's Quality Control Handbook*. McGraw-Hill, New York, 4th edition, 1988.

[15] S. T. Knox. Modeling the cost of software quality. *Digital Technical Journal*, 5(4):9–16, Fall 1993.

[16] W. L. Melo, L. C. Briand, and V. R. Basili. Measuring the impact of reuse on software quality and productivity. Technical Report CS-TR-3395, Univ. of Maryland, College Park, MD, 1995.

[17] P. Rodriguez-Dapena. Software safety certification: A multidomain problem. *IEEE Software*, 16:31–38, Jul./Aug. 1999.

[18] N. F. Schneidwind. Software metrics for quality control. In *Proc. 4th Int'l Software Metrics Symposium (METRICS '97)*, pages 127–136, Los Alamitos, Calif., Nov. 1997. IEEE CS Press.

[19] N. F. Schneidwind. Methods for assessing COTS reliability, maintainability, and availability. In *Proc. of the Int'l Conf. on Software Maintenance (ICSM '98)*, 1998.

[20] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Metrics-guided quality management for component-based software systems. In *Proc. of the 2001 Int'l Conf. on Computer Software and Applications (COMPSAC 2001)*, pages 303–308, Oct. 2001.

[21] S. Sedigh-Ali, A. Ghafoor, and R. A. Paul. Software engineering metrics for COTS-based systems. *IEEE Computer*, 34(5):44–50, May 2001.

[22] R. D. Shachter. Evaluating influence diagrams. *Operations Research*, 34(6):871–882, Nov. - Dec. 1986.

[23] S. A. Slaughter, D. E. Harter, and M. S. Krishnan. Evaluating the cost of software quality. *Comm. of the ACM*, 41(8):67–73, Aug. 1998.

[24] C. Sledge and D. Carney. Case study: Evaluating COTS products for DoD information systems. *SEI Monographs on the Use of Commercial Software in Govt. Systems*, June 1998.

[25] G. Stark, R. C. Durst, and C. W. Vowell. Using metrics in management decision making. *IEEE Software*, 11(5):42–48, May 1994.

[26] M. Tsagias and B. Kitchenham. An evaluation of the business object approach to software development. *J. Systems and Software*, 52:149–156, June 2000.

[27] J. M. Voas. The challenges of using COTS software in component-based development. *IEEE Computer*, 31(6):44–45, Jun. 1998.

[28] E. Weller. Using metrics to manage software products. *IEEE Software*, 11(5):27–33, May 1994.

Proceedings of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'03)
0-7695-1928-8/03 $17.00 © 2003 **IEEE**